2020

ILP Report

FEYSAL KETHUDA
S1842011

1. Architecture:

When designing the system, I took extra care to ensure that each part can be isolated and replaced independently of the rest of the application. The reasoning behind this is that the system is in the prototype phase and we made some assumptions that are likely to get relaxed, such as:

- Only one drone with a fixed scanning range, number of steps it can take, and only 10 degrees rotation
- The area is constrained and is small enough for us to use Euclidean distance measurement
- We connect to a single specific server address using one (unsafe) protocol

Therefore, if we need to relax any of these assumptions, introduce new constraints or optimise the algorithms we should be able to add our new requirements with minimal changes to the existing codebase.

I have therefore split the system into the following logical components:

- Web Server Connection Layer
- Distance Calculation
- Algorithms
- Core Logic

I shall explain about each component in the following subsections.

Additionally, I have created classes for some assistive functionality:

- Data containers: to group together relevant groups of data such as What3Word locations or the graphic features of sensors to be used in GeoJSON.
- Static helper services: to take care of common functionality such as writing to external files or checking if the location of the drone's next movement is within bounds and not colliding with a building.

1.1 Web Server Connection Layer:

It is always wise to isolate any interaction between our core logic and external systems, such as databases, views or external services.

This is because it can be hard otherwise to maintain the system if an external service changes the way we communicate with, or we need to change to a different service.

As such I have defined a single static web connection class where we initialise the heavyweight client object once, and every request to the server has to go through this connection layer class.

The class exposes multiple methods for the different requests that we need from the server, for example the getBuildings(), and getAirQualityData(int year, int month, int day). The classes that call these methods will either receive a JSON file that they know how to parse or null, and the details of any communication with the server is abstracted from them

1.2 Distance Calculation:

For the current requirements, we can assume that Euclidean distance is a good measure of distance. But what if we increase the flight area enough that we have to use a different Geospatial measure? Or what if we want to take other factors into consideration when calculating the path cost? **The distance calculation logic should not be closely coupled with its applications.**

This is why for this project we use a separate DistanceCalculator abstract class. Its subclasses would override the distance calculation method to any formula to get the distances between two points.

I have even defined a drone movement calculator which calculates the distance as the number of steps it would take the drone to move from one point to another. This could be used to get a more optimal tour when calculating the order to visit the sensors. I would not recommend using that as the default journey distance calculator though, because it is much slower than using Euclidean and the improvements are minimal in most cases.

The DistanceCalculator class gives us flexibility but it can make it harder for us to keep track of what distance metric each class is using. And it is not a good design if we have to change the design in multiple locations. To help resolve this issue, classes can use the DistanceCalculatorFactory which decides what distance calculator to give classes. Currently, it returns the Euclidean distance calculator as default. But very easily we can create a different calculator for heuristics or the Journey planning algorithm and apply them to the classes that use them from this distance calculator factory.

1.3 Algorithms:

The problem specification asks us to 1. find an efficient path to go through all the sensors and back to the start 2. without colliding with specified buildings and while moving in angles divisible by 10.

Therefore, it can be good to split the problem into the two subtasks mentioned above.

Since we are likely to try many different algorithms and optimisations for each of these two tasks, I decided to create JourneyStrategy and NavigationStrategy interfaces for the respective subtasks. These can then be implemented by whichever algorithms the researchers decide to test later. Classes that implement these algorithms can define helper methods to support them perform the algorithms.

In order to demonstrate how the algorithms can be used and how simple it is to swap I have also created two concrete implementations for each of them that can be easily swapped to give different results by changing one line in the main method. More details about the algorithms can be found later in this report.

1.3.1 Journey Planning:

The journey planning interface is simple; it gets an array of points that need to be visited and returns an array of the same points rearranged to minimise the distance required to travel to all of them and back.

1.3.2 Navigation:

The navigation interface defines the findPath(start, end) method which gives a navigation path between the two end points. Classes that use this interface also make use of the expand() helper method which gets the next paths for the algorithm to evaluate.

1.4 Core Logic:

- An important class in the application is the Journey class. It is given a date, start coordinates and two algorithms for journey planning & navigation. The class then records lots of information about that journey. Such as the time taken to execute and the sensors it had to visit, as well as the flight path logs in both the specified formats (txt & geoJSON).

Journey class objects are immutable, meaning that once the object is created it cannot be modified. The reasoning for this is that a journey object would not be valuable unless the algorithm is run completely and its flight path is recorded. Therefore, we run the algorithms when the object is created, and since that is expensive for runtime, we never rerun it or modify it. Instead we can only view the relevant information from that journey.

- The Coordinate class holds all information that we need from a geometric point with operations ranging from calculating angle with respect to another point to representation of the object in various ways like toString() or toMapBox() which converts it to a MapBoxSDK Point object.

- The NavigationPath class will hold the path that a drone travels across from one location to another regardless of what Navigation Algorithm is used and is helpful to conceptualise the movement of the drone.

- The Sensor class holds information about each of the air quality sensors. Each class has the coordinates of that sensor as well as methods useful for representing the readings visually through GeoJSON.

2. Documentation

App

The main class of the program.

Properties:

- Static Coordinate Start: the starting location for our journey plan

Methods:

- Private static void setup(String[]): if sufficient command line arguments are given, parses the start coordinates and port number else default values ((-3.188396, 55.944425) & 80 respectively) are used. Then sets up the server connection and gets the areas where the drone is not allowed to go from the server.

- Private static void generateFiles(JourneyStrategy, NavigationStrategy): generates the output files specified in the requirements (ones where month equals day) and puts them in the current directory.

- Private static void recordPerformance(JourneyStrategy, NavigationStrategy): records the execution time and the number of steps taken for the program to complete the journey with the algorithms in the input for all the dates in 2020 and 2021 and writes the results to a csv file named “performance.csv” in the current directory.

DistanceCalculator

Abstract class, implementations define how to calculate distance between two points.

Methods:

- public abstract Double getDistance(Coordinate p1, Coordinate p2): the basic distance calculation method from *p1* to *p2*

- public Double[][] getAllDistances(Coordinate[] coords): returns the distances between all points in the array.

- public Double getTotalDistance(Coordinate[] coords): returns the total distance from each coordinate to the next in the array.

- public Double getTourDistance(Coordinate[] coords): returns the total distance from each coordinate to the next in the array and back to the start.

EuclideanDistanceCalculator extends *DistanceCalculator*

Computes the distance assuming points are on a Euclidean plane

DroneMovementDistanceCalculator

Computes the distance based on the number of steps it takes the drone to navigate from one location to the next. For current specifications, *EuclideanDistanceCalculator* is recommended as it is much faster and the results are only marginally worse.

DistanceCalculatorFactory

Responsible for choosing which distance calculator to assign for each component.

Methods:

- public static DistanceCalculator getDefault(String calculatorType): return the appropriate distance calculator. Parameters can be replaced with enums later if many different calculators are needed to prevent confusion.

NavigationPath implements **Comparable<T>**

Path that drone takes when flying from one location to the next.

Properties:

private ArrayList<Coordinate> steps: the actual coordinates along the path.

private distanceCalculator: distance calculator for the expected path to target, defaults to "heuristic" distance calculator defined in *DistanceCalculatorFactory.*

Coordinate last: the last coordinate in the path, stored to speed up getting heuristic cost.

Coordinate target: where the path needs to get to.

Constructors:

- NavigationPath(Coordinate target, ArrayList<Coordinate>): to create a path from pre-existing coordinate list
- NavigationPath(Coordinate target) for a new path

Methods:

- public void add(Coordinate coord): add coordinate to path
- public Double getPathCost(): return distance travelled so far + distance to target
- public Double getDistanceToTarget(): returns distance to target with respect to the distance calculator specified in the class variables.
- public int compareTo(NavigationPath otherPath): Overriden from *Comparable<>*, compares paths based on path cost.

NavigationStrategy

Interface, implementations define how a drone goes from one point to the next.

Methods:

- Coordinate[] findPath(Coordinate start, Coordinate finish): return the coordinates passed on the way from *start* to *finish*
- ArrayList<NavigationPath> expand(NavigationPath path): returns an arraylist of copies of the current NavigationPath each with a point where the drone can move to next appended to its end.

AStarNavigationStrategy Implements NavigationStrategy

Properties:

- distanceCalculator: the distance calculator used for navigation, defaults to the "navigation" distance calculator defined in *DistanceCalculatorFactory.*

Methods:

- Coordinate[] findPath(Coordinate start, Coordinate finish): overridden from NavigationStrategy, uses the modified A Star algorithm discussed in the Algorithms section.

- private ArrayList<NavigationPath> expand(NavigationPath path): overridden from NavigationStrategy, returns all possible expansions.

private ArrayList<NavigationPath> expand(int toTake, NavigationPath path): similar to the one parameter version, but returns the smallest *toTake* paths instead of all possible paths

<u>BestFirstNavigationStrategy</u> Implements *NavigationStrategy*

Properties:

- distanceCalculator: the distance calculator used for navigation, defaults to "navigation" calculator in *DistanceCalculatorFactory.*

Methods:

- Coordinate[] findPath(Coordinate start, Coordinate finish): overridden from NavigationStrategy, uses the *Greedy Best First* algorithm discussed in the Algorithms section. NOTE: Using this algorithm may result in no path being found. For a complete algorithm please use A Star or other navigation strategies.

- public ArrayList<NavigationPath> expand(NavigationPath path): returns an arraylist containing only the shortest possible expansion to the current path.

<u>JourneyStrategy</u>

Interface, implementations define the order in which the drone visits the sensors.

Methods:

- Coordinate[] planJourney(Coordinate[]): returns the sorted order of coordinates to visit.

<u>TwoOptJourneyStrategy</u> implements *JourneyStrategy*

constant ITERATION_LIMIT = 800: times to repeat swaps without finding improvements.

- distanceCalculator: the distance calculator used for journey planning, defaults to "journey" calculator in *DistanceCalculatorFactory*

Methods:

- public Coordinate[] planJourney(Coordinate[]): overridden from *JourneyStrategy,* performs 2-opt algorithm, which is discussed in more detail in the Algorithm section of this report.

- private void TwoOptSwap(int i, int k, Coordinate[] tour, Coordinate[] newTour): helper function that performs the swap on indices i and k in a tour and returns the swapped version.

<u>GreedyJourneyStrategy</u> implements *JourneyStrategy*

- distanceCalculator: the distance calculator used for journey planning, defaults to "journey" calculator in *DistanceCalculatorFactory*

Methods:

- public planJourney(Coordinate[] tour): overridden from *JourneyStrategy,* uses helper to calculate the Greedy order of the sensors to visit then returns the coordinates corresponding to the tour indices. Recommended when number of sensors is too large for 2-opt and path efficiency is not a priority.

- private Integer[] findMinRoute(Double[][] distances): performs Greedy algorithm, which goes to the closest unvisited neighbour from current location.

Coordinate

The geographic representation of a point that we use in this project.

Properties:

- private Double lng: longitude (horizontal or x value)

- private Double lat: latitude (vertical or y value)

Methods:

- public Coordinate getPointAt(double radius, int degree): gets the point in the direction of *degree (in degrees)* that is *radius* away from this object.

- public double getAngle(Coordinate target): gets angle between this object and the target

- public Point toMapBox(): returns the MapBoxSDK Point representation of this object

- hashcode() and equals() to make the object hashable. Which is useful when we want to use it as a HashMap key

Sensor

The model class of the sensor, native types are parsed from server and are used to create the object properties.

Properties:

- private (get) String location: the location of the sensor represented as What3Word.

- private double battery: sensor battery, reading is unreliable if less than 10.0

- private String reading: the pollution reading at sensor

- private What3Word word: contains the What3Word of the sensor, created from its location property, see What3Word class for more details.

- private ReadingProperties readingProperties: graphical properties of the sensor, see ReadingProperties for more details.

Methods:

- public Feature toMapBox(): returns the MapBoxSDK Feature representation of this object.

- public Coordinate getCoordinates(): returns the longitude, latitude coordinates of the sensor.

What3Word Contains the What3Words *word* and the longitude, latitude *coordinates* of the sensor.

ReadingProperties contains the graphical properties for the sensor.

private (get) String color: rgb string of the reading.

private (get) String symbol: symbol name for geoJSON private (get) String colorName: color name to be used in geoJSON

ProjectConstants constants given in the project specification

double STEP_SIZE: one drone move step size.

double SCAN_RANGE: how close can the drone be to scan sensor

Int MAX_STEPS: maximum number of steps a drone can take

double ROTATION_ANGLE: how much can the drone rotate in angular degrees

double[4] BOUNDARIES: top, bottom, left, right boundaries

String MARKER_SIZE: for GeoJSON representation

Building encapsulates a building that the drone can't pass through

- private List<Coordinate> coordinates: corners of the building

Methods:

- boolean passedBy(Line2D path): checks if a given path passes through a building

LocationChecker checks if a location is valid for the drone to move to

- private static List<Building> noFlyAreas: list of areas that a building is not allowed to fly through

Methods:

- public static void setup(): initialises the buildings from data retrieved from the server

- public static boolean isValidPath(Coordinate p1, Coordinate p2): makes sure that the path between two points does not intersect any buildings and that the destination is within bounds

- private static boolean isColliding(Coordinate p1, Coordinate p2): does not touch any building

- private static boolean isWithinBounds(Coordinate p): not out of bounds

WebServerConnection handles all external communication with server

public static void setPort(int port): specify the port to communicate with server

private static final HttpClient CLIENT: the java client object for this project

Methods:

- private static HttpResponse<String> makeRequest(String subDirectory): makes the request to the server with the given sub directory, returns JSON body if success else returns null

- public static String getBuildings(): get building data from server. Returns JSON string. used for initialising Building objects.

- public static String getAirQualityData(int year, int month, int day): get Air Quality data for a specific date from the server. Returns JSON string. used for initialising Journey objects.

- public static String getWords(String[] words): gets the word coordinates object from the server, used for initialising What3Word objects.

Journey Records data about the flight on a specific date, immutable (see architecture section).

Properties:

- ints day, month, year: date of journey
- ArrayList<Sensor> sensors: sensors that need to be visited
- HashMap<Coordinate, Sensor> sensorLocations: locations where the sensors are and their corresponding sensors
- Coordinate[] orderedJourney: the tour that the drone has to go through
- ArrayList<Coordinate> visited: points that the drone flew through
- int visitedSensorsCount: how many sensors we have scanned in order
- int journeyLength: number of steps the drone took to complete the journey
- Duration executionTime: execution time of the algorithms
- String movements: log of drone movement
- String sensorReadings: drone movement in GeoJSON form

Methods:

- private void parseSensors(): get sensor data from server
- private void sensorLocations(): initialise object's sensorLocations hashmap
- private String recordJourneyPath(navigationStrategy): perform the flight for this date.
- private String recordLocalNavigation(start, end, int nextSensorIndex, NavigationStrategy): helper function that performs navigation from one location to the next. *nextSensorIndex* parameter is used to get the sensor that is at the target.
- private String recordReadings(): records the flight path and readings in GeoJSON format

OutputWriter writes results to external files

Methods:

- public static void write(String content, String fileName): writes *content* String to a file and names it *filename*
- public static void writeReadingsFile(Journey journey): writes the GeoJSON readings for the specified *Journey* in a file with the specified name format.
- public static void writeFlightPathFile(Journey journey): writes the drone flight path for the specified *Journey* in a file with the specified name format.

3. Algorithm:

The drone employs two different algorithms to sort the sensors and to navigate from one sensor to the next respectively.

3.1 Journey Planning Algorithm:

the journey planning problem is in essence a travelling salesman problem where the drone starts at one point and has to visit all sensor locations and return to the start while trying to keep the path as short as possible.

Currently, there is no known method to get the optimal solution other than using brute force which has a complexity of O(n!), which is infeasible in our case. Luckily, the drone has enough battery to finish the journey with a slightly suboptimal tour.

The main journey planning algorithm that the drone uses is the 2-opt local search heuristic. The main idea of this heuristic is to take a tour that crosses over itself and reorder it so that it does not.

Not much is known about the theoretical aspect of the heuristic such as its exact complexity, but in practice it performs really well on points on a 2-dimensional Euclidean plane, with respect to both execution time and approximation of the result to the optimal tour.

The algorithm may perform worse if we start using non-Euclidean distance calculators, for example ones taking into account wind movement or flying at slower speeds in some areas. However, it works well for our current requirements and the project is designed in a way that makes it easy to replace this algorithm with a more suitable one.

For our case, we keep trying to improve the results by flipping every two points in the tour then checking if we get a better tour, and only stop when the optimal tour doesn't change for an arbitrary number of iterations (default is 800).

After the order in which the sensors need to be visited is found, the drone then iterates through the tour navigating from one sensor to the next using its navigation algorithm.

3.2 Navigation Algorithm:

For the navigation between two points the drone employs an implementation of the A star algorithm. The algorithm takes a starting point then while there are still unexplored points on the frontier it calculates the next potential steps from the current best path, these are the points that satisfy the following conditions:

- are 0.0003 degrees away from the current point.
- are at angles divisible by ten with relation to the point.
- are within bounds and the path there does not intersect with any of the areas in the no-fly zone as obtained from the server.

The algorithm then appends each of these points to a copy of the path and places the paths in a priority queue which guarantees sorted ordering of the paths, O(log(n)) insertion and constant time retrieval of the path with smallest distance travelled so far. where n is the number of paths in the heap.

The current implementation uses a Java PriorityQueue and a custom comparator for paths to get the intended functionality for frontier paths that need to be explored.

A-Star is both complete and optimal, meaning it is guaranteed to find a solution if one exists and the path it finds is guaranteed to be an optimal path. And the heuristic it uses (Euclidean distance) is admissible, meaning that it never overestimates the cost of a path. This makes A star good for navigating the path between any two nodes.

A Star has a worst-case performance complexity of O(b^d) where b is the number of next possible moves (up to 36 each step) and d is the depth of the graph (number of steps to reach target). A Star also stores all the frontier nodes in the queue, which leads to massive memory usage O(b^d). if the search space is big enough the program is likely to run out of memory and crash. In our case, this usually happens if the path from the start to the end nodes is more than 10 steps.

In order to work around this issue, we can use the fact that our drone does not encounter many obstacles and usually travels in one direction. If the distance is too large, we can prune the number of branches to explore by taking the most promising paths only. The closer we get to the target the more branches we consider. The number of branches I take is determined using the following formula:

$$n = floor(18 / s) + 1$$

s here is the estimated number of steps to target, we get it using the heuristic distance to target divided by the step size. Thus, the closer we are to the target the more options we try to get as close as possible to the target.

3.3 Performance:

After trying the algorithm on all every day in 2020 and 2021. I recorded the running times and the number of steps taken in each journey. The results are saved in “performance.xlsx” in the submitted source code and the csv can be generated in the code to get the results on your machine.

I have used the starting point that was provided for the “ilp-results”. I found that the navigation part was the main bottleneck, and it took much longer to finish than the journey planning stage. We have to also keep in mind that the execution time will vary depending on the machine running the program, but checking it helps to give us an idea of where the algorithm struggles and where it performs well.

I found that there are several maps with the same set of sensors where the drone will take the same number of steps to complete the journey. The longest of these was the map that can be found on 20/02/2020. It takes 102 steps to finish the journey and the path is shown in figure 1:

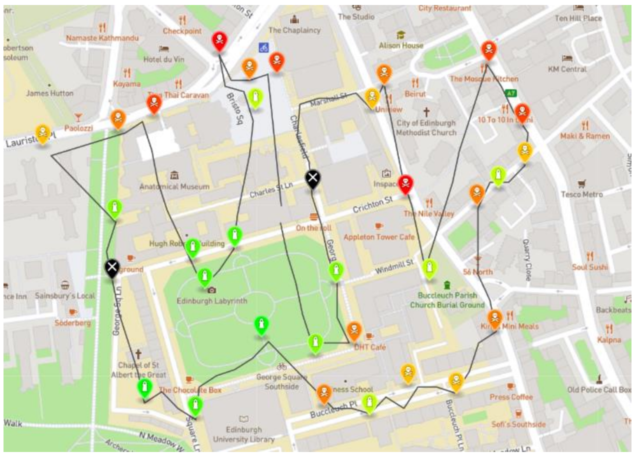

Figure 1: Longest Journey (102 steps) on 20/02/2020

The shortest journey was when all the sensors were placed in George Square. Resulting in a journey of 47 steps, as can be seen from figure 2:

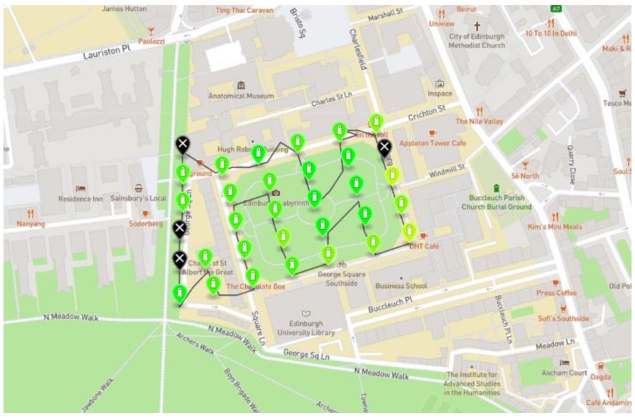

Figure 2: Shortest Journey (47 steps) on 01/04/2020

The A Star algorithm gives a good navigation path but in some special cases takes a long time to finish running. The longest runtime was just over 15 seconds, and the worst 8 journeys taking between 10.7 and 15.1 were all on the same map shown in the following figure:

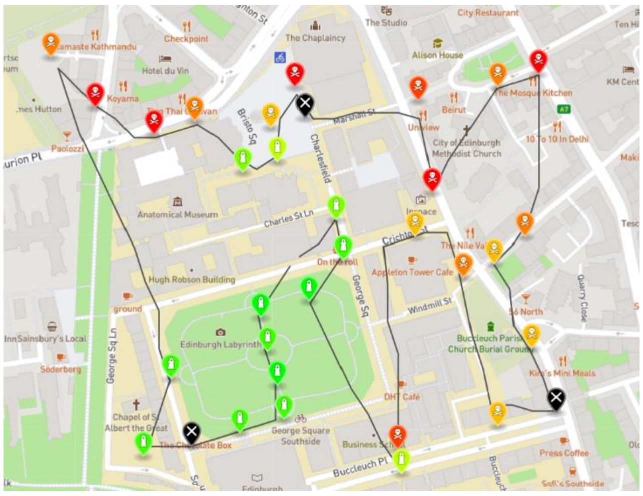

Figure 3: Longest Runtime (15.14 seconds) on 22/12/2021, which took 89 steps.

The cause of the long running time is likely the very long line on the left. When I moved the starting point to the left of the Anatomical Museum, running time dropped to 0.87 seconds. Thus if the researchers are able to choose where the drone should start, they should place it on the path of the longest line to increase runtime speed, however number of steps taken may be less optimal, so they will have to choose what to prioritise.

The total execution time of all 732 maps was just over 320 seconds (5 min 20 seconds), which gives a mean runtime of 0.44 seconds per journey. The mean number of steps taken to complete a journey was 89. This will vary based in starting location but is still quite a bit smaller than the limit.

I found that the correlation between running time and number of steps in the journey is very weak, with the correlation coefficient being 0.0794. Additionally, I noticed that the dates which took the longest to compute were between average and slightly above average number of moves.